4 Using the development tools

This chapter covers

- Using command-line tools to create an ASP.NET Core project
- Adding code and content to a project
- Building and running an ASP.NET Core project
- Using the hot reload feature
- Installing NuGet packages
- Installing tool packages
- Installing client-side packages
- Using the debugger

In this chapter, I introduce the tools that Microsoft provides for ASP.NET Core development and that are used throughout this book.

Unlike earlier editions of this book, I rely on the command-line tools provided by the .NET SDK and additional tool packages that Microsoft publishes. In part, I have done this to help ensure you get the expected results from the examples but also because the command-line tools provide access to all the features required for ASP.NET Core development, regardless of which editor/IDE you have chosen.

Visual Studio—and, to a lesser extent, Visual Studio Code—offers access to some of the tools through user interfaces, which I describe in this chapter, but Visual

Studio and Visual Studio Code don't support all the features that are required for ASP.NET Core development, so there are times that using the command line is inevitable.

As ASP.NET Core has evolved, I have gradually moved to using just the command-line tools, except for when I need to use a debugger (although, as I explain later in the chapter, this is a rare requirement). Your preferences may differ, especially if you are used to working entirely within an IDE, but my suggestion is to give the command-line tools a go. They are simple, concise, and predictable, which cannot be said for all the equivalent functionality provided by Visual Studio and Visual Studio Code. Table 4.1 provides a guide to the chapter.

Table 4.1 Chapter guide

Problem	Solution	Listing
Creating a project	Use the `dotnet new` commands.	1–3
Building and running projects	Use the `dotnet build` and `dotnet run` commands.	4–10
Adding packages to a project	Use the `dotnet add package` command.	11, 12
Installing tool commands	Use the `dotnet tool` command.	14, 15
Managing client-side packages	Use the `libman` command or the Visual Studio client-side package manager.	16–19

4.1 Creating ASP.NET Core projects

The .NET SDK includes a set of command-line tools for creating, managing, building, and running projects. Visual Studio provides integrated support for some of these tasks, but if you are using Visual Studio Code, then the command line is the only option.

I use the command-line tools throughout this book because they are simple and concise. The Visual Studio integrated support is awkward and makes it easy to unintentionally create a project with the wrong configuration, as the volume of emails from confused readers of earlier editions of this book has demonstrated.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/manningbooks/pro-asp.net-core-7. See chapter 1 for how to get help if you have problems running the examples.

4.1.1 Creating a project using the command line

The `dotnet` command provides access to the .NET command-line features. The `dotnet new` command is used to create a new project, configuration file, or solution file. To see the list of templates available for creating new items, open a PowerShell command prompt and run the command shown in listing 4.1.

Listing 4.1 Listing the .NET templates

```
dotnet new --list
```

Each template has a short name that makes it easier to use. There are many templates available, but table 4.2 describes the ones that are most useful for creating ASP.NET Core projects.

Table 4.2 Useful ASP.NET Core project templates

Name	Description
`web`	This template creates a project that is set up with the minimum code and content required for ASP.NET Core development. This is the template I use for most of the chapters in this book.
`mvc`	This template creates an ASP.NET Core project configured to use the MVC Framework.
`webapp`	This template creates an ASP.NET Core project configured to use Razor Pages.
`blazorserver`	This template creates an ASP.NET Core project configured to use Blazor Server.
`angular`	This template creates an ASP.NET Core project that contains client-side features using the Angular JavaScript framework.
`react`	This template creates an ASP.NET Core project that contains client-side features using the React JavaScript framework.
`reactredux`	This template creates an ASP.NET Core project that contains client-side features using the React JavaScript framework and the popular Redux library.

There are also templates that create commonly required files used to configure projects, as described in table 4.3.

Understanding the limitations of project templates

The project templates described in table 4.2 are intended to help jump-start development by taking care of basic configuration settings and adding placeholder content.

These templates can give you a sense of rapid progress, but they contain assumptions about how a project should be configured and developed. If you don't understand the impact of those assumptions, you won't be able to get the results you require for the specific demands of your project.

The `web` template creates a project with the minimum configuration required for ASP.NET Core development. This is the project template I use for most of the examples in this book so that I can explain how each feature is configured and how the features can be used together.

Once you understand how ASP.NET Core works, the other project templates can be useful because you will know how to adapt them to your needs. But, while you are learning, I recommend sticking to the `web` template, even though it can take a little more effort to get results.

Table 4.3 The configuration item templates

Name	Description
`globaljson`	This template adds a `global.json` file to a project, specifying the version of .NET that will be used.
`sln`	This template creates a solution file, which is used to group multiple projects and is commonly used by Visual Studio. The solution file is populated with the `dotnet sln add` command, as shown in listing 4.2.
`gitignore`	This template creates a `.gitignore` file that excludes unwanted items from Git source control.

To create a project, open a new PowerShell command prompt and run the commands shown in listing 4.2.

Listing 4.2 Creating a new project

```
dotnet new globaljson --sdk-version 7.0.100 --output MySolution/MyProject
dotnet new web --no-https --output MySolution/MyProject --framework net7.0
dotnet new sln -o MySolution
dotnet sln MySolution add MySolution/MyProject
```

The first command creates a `MySolution/MyProject` folder that contains a `global.json` file, which specifies that the project will use .NET version 7. The top-level folder, named `MySolution`, is used to group multiple projects. The nested `MyProject` folder will contain a single project.

I use the `globaljson` template to help ensure you get the expected results when following the examples in this book. Microsoft is good at ensuring backward compatibility with .NET releases, but breaking changes do occur, and it is a good idea to add a `global.json` file to projects so that everyone in the development team is using the same version.

The second command creates the project using the `web` template, which I use for most of the examples in this book. As noted in table 4.3, this template creates a project with the minimum content required for ASP.NET Core development. Each template has its own set of arguments that influence the project that is created. The `--no-https` argument creates a project without support for HTTPS. (I explain how to use HTTPS in chapter 16.) The `--framework` argument selects the .NET runtime that will be used for the project.

The other commands create a solution file that references the new project. Solution files are a convenient way of opening multiple related files at the same time. A `MySolution.sln` file is created in the `MySolution` folder, and opening this file in Visual Studio will load the project created with the web template. This is not essential, but it stops Visual Studio from prompting you to create the file when you exit the code editor.

OPENING THE PROJECT

To open the project, start Visual Studio, select Open a Project or Solution, and open the `MySolution.sln` file in the `MySolution` folder. Visual Studio will open the solution file, discover the reference to the project that was added by the final command in listing 4.2, and open the project as well.

Visual Studio Code works differently. Start Visual Studio Code, select File > Open Folder, and navigate to the `MySolution` folder. Click Select Folder, and Visual Studio Code will open the project.

Although Visual Studio Code and Visual Studio are working with the same project, each displays the contents differently. Visual Studio Code shows you a simple list of files, ordered alphabetically, as shown on the left of figure 4.1. Visual Studio hides some files and nests others within related file items, as shown on the right of figure 4.1.

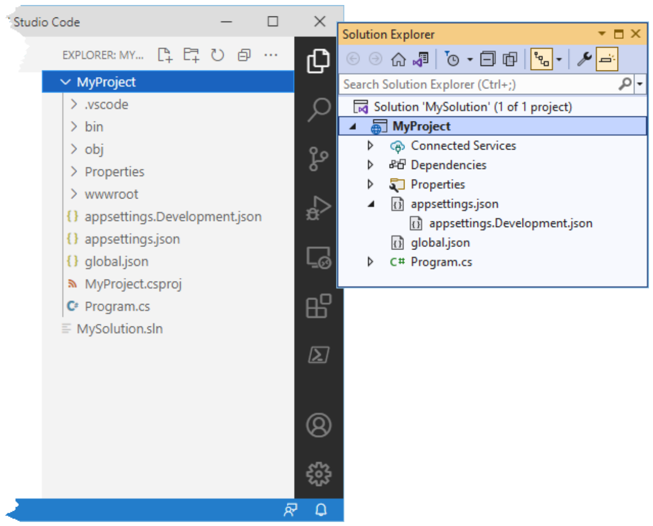


Figure 4.1 Opening a project in Visual Studio Code and Visual Studio

There are buttons at the top of the Visual Studio Solution Explorer that disable file nesting and show the hidden items in the project. When you open a project for the first time in Visual Studio Code, you may be prompted to add assets for building and debugging the project. Click the Yes button.

4.2 Adding code and content to projects

If you are using Visual Studio Code, then you add items to the project by right-clicking the folder that should contain the file and selecting New File from the pop-up menu (or selecting New Folder if you are adding a folder).

NOTE You are responsible for ensuring that the file extension matches the type of item you want to add; for example, an HTML file must be added with the `.html` extension. I give the complete file name and the name of the containing folder for every item added to a project throughout this book, so you will always know exactly what files you need to add.

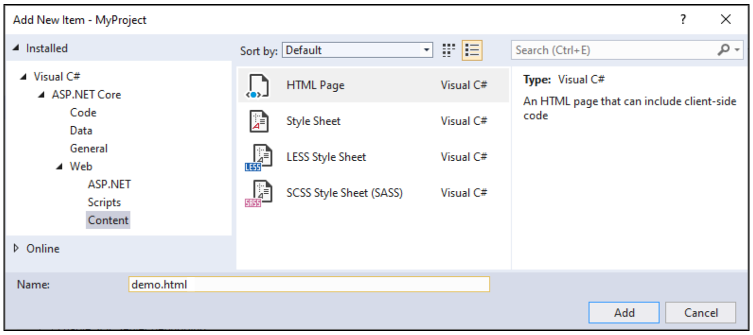
Right-click the My Project item in the list of files and select New Folder from the pop-up menu. Set the name to `wwwroot`, which is where static content is stored in ASP.NET Core projects. Press Enter, and a folder named `wwwroot` will be added to the project. Right-click the new `wwwroot` folder, select New File, and set the name to `demo.html`. Press Enter to create the HTML file and add the content shown in listing 4.3.

Listing 4.3 The contents of the demo.html file in the wwwroot folder

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <h3>HTML File from MyProject</h3>
</body>
</html>
```

Visual Studio provides a more comprehensive approach that can be helpful, but only when used selectively. To create a folder, right-click the MyProject item in the Solution Explorer and select Add > New Folder from the pop-up menu. Set the name of the new item to `wwwroot` and press Enter; Visual Studio will create the folder.

Right-click the new wwwroot item in the Solution Explorer and select Add > New Item from the pop-up menu. Visual Studio will present you with an extensive selection of templates for adding items to the project. These templates can be searched using the text field in the top-right corner of the window or filtered using the categories on the left of the window. The item template for an HTML file is named HTML Page, as shown in figure 4.2.



Sort by: Default

HTML Page	Visual C#
Style Sheet	Visual C#
LESS Style Sheet	Visual C#
SCSS Style Sheet (SASS)	Visual C#

Type: Visual C#
An HTML page that can include client-side code

Name: demo.html

Figure 4.2 Adding an item to the example project

Enter `demo.html` in the Name field, click the Add button to create the new file, and replace the contents with the element shown in listing 4.3. (If you omit the file extension, Visual Studio will add it for you based on the item template you have selected. If you entered just `demo` into the Name field when you created the file, Visual Studio would have created a file with the `.html` extension because you had selected the HTML Page item template.)

4.2.1 Understanding item scaffolding

The item templates presented by Visual Studio can be useful, especially for C# classes where it sets the namespace and class name automatically. But Visual Studio also provides *scaffolded items*, which I recommend against using. The Add > New Scaffolded Item leads to a selection of items that guide you through a process to add more complex items. Visual Studio will also offer individual scaffolded items based on the name of the folder that you are adding an item to. For example, if you right-click a folder named `Views`, Visual Studio will helpfully add scaffolded items to the top of the menu, as shown in figure 4.3.

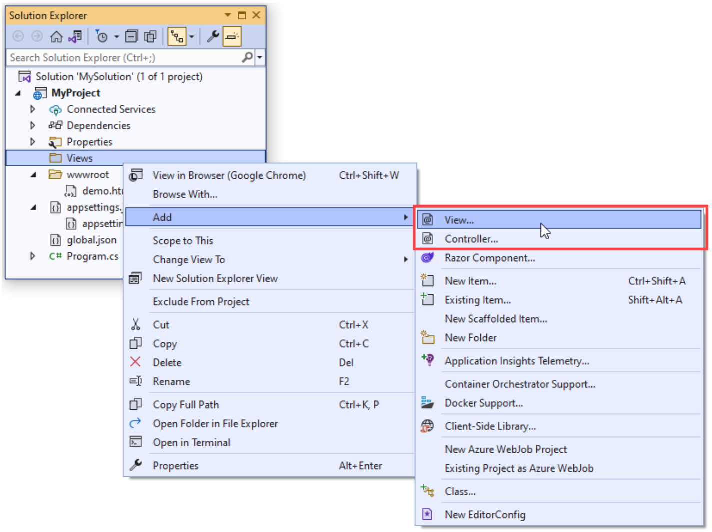


Figure 4.3 Scaffolded items in the Add menu

The `View` and `Controller` items are scaffolded, and selecting them will present you with choices that determine the content of the items you create.

Just like the project templates, I recommend against using scaffolded items, at least until you understand the content they create. In this book, I use only the Add > New Item menu for the examples and change the placeholder content immediately.

4.3 Building and running projects

The simplest way to build and run a project is to use the command-line tools. To prepare, add the statement shown in listing 4.4 to the `Program.cs` class file in the `MyProject` folder.

Listing 4.4 Adding a statement in the Program.cs file in the MyProject folder

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.UseStaticFiles();

app.Run();
```

This statement adds support for responding to HTTP requests with static content in the `wwwroot` folder, such as the HTML file created in the previous section. (I explain this feature in more detail in chapter 15.)

Next, set the HTTP port that ASP.NET Core will use to receive HTTP requests, as shown in listing 4.5.

Listing 4.5 Setting the HTTP port in the launchSettings.json file in the Properties folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "MyProject": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

To build the example project, run the command shown in listing 4.6 in the `MyProject` folder.

Listing 4.6 Building the project

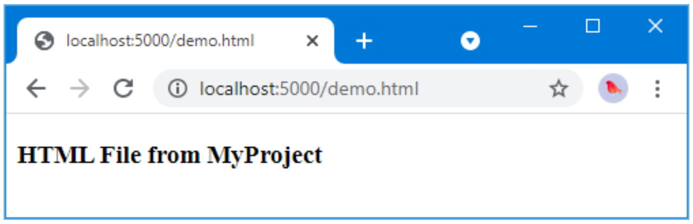
```
dotnet build
```

You can build and run the project in a single step by running the command shown in listing 4.7 in the `MyProject` folder.

Listing 4.7 Building and running the project

```
dotnet run
```

The compiler will build the project and then start the integrated ASP.NET Core HTTP server to listen for HTTP requests on port 5000. You can see the contents of the static HTML file added to the project earlier in the chapter by opening a new browser window and requesting http://localhost:5000/demo.html, which produces the response shown in figure 4.4.



HTML File from MyProject

Figure 4.4 Running the example application

4.3.1 Using the hot reload feature

.NET has an integrated hot reload feature, which compiles and applies updates to applications on the fly. For ASP.NET Core applications, this means that changes to the project are automatically reflected in the browser without having to manually stop the ASP.NET Core application and use the `dotnet run` command. Use Control+C to stop ASP.NET Core if the application is still running from the previous section and run the command shown in listing 4.8 in the `MyProject` folder.

Listing 4.8 Starting the application with hot reload

```
dotnet watch
```

The `dotnet watch` command opens a new browser window, which it does to ensure that the browser loads a small piece of JavaScript that opens an HTTP connection to the server that is used to handle reloading. (The new browser window can be disabled by setting the `launchBrowser` property shown in listing 4.5 to `false`, but you will have to perform a manual reload the first time you start or restart ASP.NET Core.) Use the browser to request http://localhost:5000/demo.html, and you will see the output shown on the left of figure 4.5.

The `dotnet watch` command monitors the project for changes. When a change is detected, the project is automatically recompiled, and the browser is reloaded. To see this process in action, make the change shown in listing 4.9 to the `demo.html` file in the `wwwroot` folder.

Listing 4.9 Changing the message in the demo.html file in the wwwroot folder

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <h3>New Message</h3>
</body>
</html>
```

When you save the changes to the HTML file, the `dotnet watch` tool will detect the change and automatically update the browser, as shown in figure 4.5.

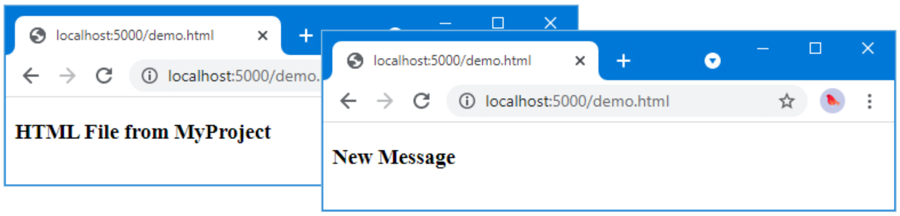


Figure 4.5 The hot reload feature

The `dotnet watch` command is a clever feat of engineering, and it has good support for ASP.NET Core applications, allowing changes to be easily applied. But not all changes can be handled with a hot reload.

If you are using Visual Studio, right-click the MyProject item in the Solution Explorer, select Add > Class from the pop-up menu, and set the name of the new class file to `MyClass.cs`. When Visual Studio opens the file for editing, change the namespace as shown in listing 4.10.

Listing 4.10 Changing a namespace in the MyClass.cs file in the MyProject folder

```
namespace MyProject.MyNamespace {
    public class MyClass {
    }
}
```

If you are using Visual Studio Code, add a file named `MyClass.cs` to the `MyProject` folder with the content shown in listing 4.10.

Regardless of which editor you use, you will see output similar to the following when you save the class file:

```
watch : File changed: C:\MySolution\MyProject\MyClass.cs.
watch : Unable to apply hot reload because of a rude edit.
```

There are some changes that the `dotnet watch` command can't handle with a hot reload and the application is restarted instead. You may be prompted to accept the restart. The restart has little effect on the example application, but it means that the application state is lost, which can be frustrating when working on real projects.

But even though it isn't perfect, the hot reload feature is useful, especially when it comes to iterative adjustments to the HTML an application produces. I don't use it in most of the chapters in this book because the examples require many changes that are not handled with hot reloads and that can prevent changes from taking effect, but I do use it for my own non-book related development projects.

4.4 Managing packages

Most projects require additional features beyond those set up by the project templates, such as support for accessing databases or for making HTTP requests, neither of which is included in the standard ASP.NET Core packages added to the project by the template used to create the example project. In the sections that follow, I describe the tools available to manage the different types of packages that are used in ASP.NET Core development.

4.4.1 Managing NuGet packages

.NET packages are added to a project with the `dotnet add package` command. Use a PowerShell command prompt to run the command shown in listing 4.11 in the `MyProject` folder to add a package to the example project.

Listing 4.11 Adding a package to the example project

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 7.0.0
```

This command installs version 7.0.0 of the `Microsoft.EntityFrameworkCore.SqlServer` package. The package repository for .NET projects is `nuget.org`, where you can search for the package and see the versions available. The package installed in listing 4.11, for example, is described at https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer/7.0.0. You can see the packages installed in a project by running the command shown in listing 4.12.

> **TIP** The project file—which is the file with the `.csproj` extension—is used to keep track of the packages added to a project. You can examine this file by opening it for editing in Visual Studio Code or by right-clicking the project item in the Visual Studio Solution Explorer and selecting Edit Project File from the pop-up menu.

Listing 4.12 Listing the packages in a project

```
dotnet list package
```

This command produces the following output when it is run in the `MyProject` folder, showing the package added in listing 4.11:

```
Project 'MyProject' has the following package references
   [net7.0]:
   Top-level Package                              Requested    Resolved
   > Microsoft.EntityFrameworkCore.SqlServer      7.0.0        7.0.0
```

Packages are removed with the `dotnet remove package` command. To remove the package from the example project, run the command shown in listing 4.13 in the `MyProject` folder.

Listing 4.13 Removing a package from the example project

```
dotnet remove package Microsoft.EntityFrameworkCore.SqlServer
```

4.4.2 Managing tool packages

Tool packages install commands that can be used from the command line to perform operations on .NET projects. One common example is the Entity Framework Core tools package that installs commands that are used to manage databases in ASP.NET Core projects. Tool packages are managed using the `dotnet tool` command. To install the Entity Framework Core tools package, run the commands shown in listing 4.14.

Listing 4.14 Installing a tool package

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 7.0.0
```

The first command removes the `dotnet-ef` package, which is named `dotnet-ef`. This command will produce an error if the package has not already been installed, but it is a good idea to remove existing versions before installing a package. The `dotnet tool install` command installs version 7.0.0 of the `dotnet-ef` package, which is the version I use in this book. The commands installed by tool packages are used through the `dotnet` command. To test the package installed in listing 4.14, run the command shown in listing 4.15 in the `MyProject` folder.

> **TIP** The `--global` arguments in listing 4.14 mean the package is installed for global use and not just for a specific project. You can install tool packages into just one project, in which case the command is accessed with `dotnet tool run <command>`. The tools I use in this book are all installed globally.

Listing 4.15 Running a tool package command

```
dotnet ef --help
```

The commands added by this tool package are accessed using `dotnet ef`, and you will see examples in later chapters that rely on these commands.

4.4.3 Managing client-side packages

Client-side packages contain content that is delivered to the client, such as images, CSS stylesheets, JavaScript files, and static HTML. Client-side packages are added to ASP.NET Core using the Library Manager (LibMan) tool. To install the LibMan tool package, run the commands shown in listing 4.16.

Listing 4.16 Installing the LibMan tool package

```
dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli
dotnet tool install --global Microsoft.Web.LibraryManager.Cli
    --version 2.1.175
```

These commands remove any existing LibMan package and install the version that is used throughout this book. The next step is to initialize the project, which creates the file that LibMan uses to keep track of the client packages it installs. Run the command shown in listing 4.17 in the `MyProject` folder to initialize the example project.

Listing 4.17 Initializing the example project

```
libman init -p cdnjs
```

LibMan can download packages from different repositories. The `-p` argument in listing 4.17 specifies the repository at https://cdnjs.com, which is the most widely used. Once the project is initialized, client-side packages can be installed. To install the Bootstrap CSS framework that I use to style HTML content throughout this book, run the command shown in listing 4.18 in the `MyProject` folder.

Listing 4.18 Installing the Bootstrap CSS framework

```
libman install bootstrap@5.2.3 -d wwwroot/lib/bootstrap
```

The command installs version 5.2.3 of the Bootstrap package, which is known by the name `bootstrap` on the CDNJS repository. The `-d` argument specifies the location into which the package is installed. The convention in ASP.NET Core projects is to install client-side packages into the `wwwroot/lib` folder.

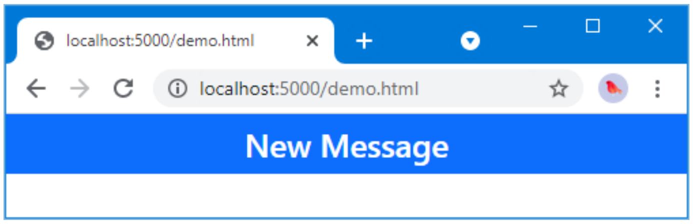
Once the package has been installed, add the classes shown in listing 4.19 to the elements in the `demo.html` file. This is how the features provided by the Bootstrap package are applied.

NOTE I don't get into the details of using the Bootstrap CSS framework in this book. See https://getbootstrap.com for the Bootstrap documentation.

Listing 4.19 Applying Bootstrap classes in the demo.html file in the wwwroot folder

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
    <link href="/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <h3 class="bg-primary text-white text-center p-2">New Message</h3>
</body>
</html>
```

Start ASP.NET Core and request http://localhost:5000/demo.html, and you will see the styled content shown in figure 4.6.


localhost:5000/demo.html

New Message

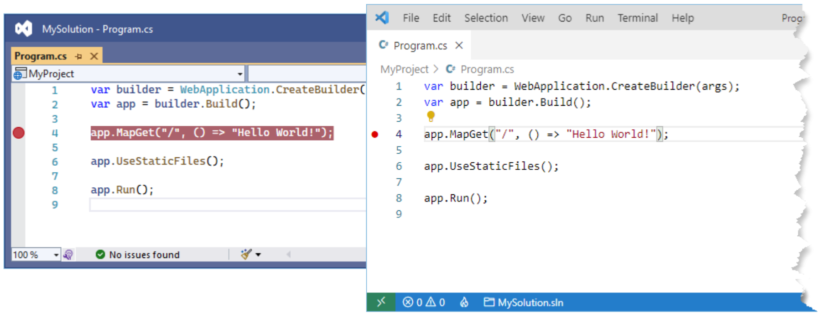
Figure 4.6 Using a client-side package

4.5 *Debugging projects*

Visual Studio and Visual Studio Code both provide debuggers that can be used to control and inspect the execution of an ASP.NET Core application. Open the `Program.cs` file in the `MyProject` folder, and click this statement in the code editor:

```
...
app.MapGet("/", () => "Hello World!");
...
```

Select Debug > Toggle Breakpoint in Visual Studio or select Run > Toggle Breakpoint in Visual Studio Code. A breakpoint is shown as a red dot alongside the code statement, as shown in figure 4.7, and will interrupt execution and pass control to the user.



```
var builder = WebApplication.CreateBuilder(
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.UseStaticFiles();

app.Run();
```

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.UseStaticFiles();

app.Run();
```

Figure 4.7 Setting a breakpoint

Start the project by selecting Debug > Start Debugging in Visual Studio or selecting Run > Start Debugging in Visual Studio Code. (Choose .NET if Visual Studio Code prompts you to select an environment and then select the Start Debugging menu item again.)

The application will be started and continue normally until the statement to which the breakpoint is reached, at which point execution is halted. Execution can be controlled using the Debug or Run menu or the controls that Visual Studio and Visual Studio Code display. Both debuggers are packed with features—more so if you have a paid-for version of Visual Studio—and I don't describe them in depth in this book. The Visual Studio debugger is described at https://docs.microsoft.com/en-us/visual studio/debugger, and the Visual Studio Code debugger is described at https://code .visualstudio.com/docs/editor/debugging.

How I debug my code

Debuggers are powerful tools, but I rarely use them. In most situations, I prefer to add `Console.WriteLine` statements to my code to figure out what is going on, which I can easily do because I use the `dotnet run` command to run my projects from the command line. This is a rudimentary approach that works for me, not least because most of the errors in my code tend to be where statements are not being called because a condition in an `if` statement isn't effective. If I want to examine an object in detail, I tend to serialize it to JSON and pass the result to the `WriteLine` method.

This may seem like madness if you are a dedicated user of the debugger, but it has the advantage of being quick and simple. When I am trying to figure out why code isn't working, I want to explore and iterate quickly, and I find the amount of time taken to start the debugger to be a barrier. My approach is also reliable. The Visual Studio and Visual Studio Code debuggers are sophisticated, but they are not always entirely predictable, and .NET and ASP.NET Core change too quickly for the debugger features to have entirely settled down. When I am utterly confused by the behavior of some code, I want the simplest possible diagnostic tool, and that, for me, is a message written to the console.

I am not suggesting that this is the approach you should use, but it can be a good place to start when you are not getting the results you expect and you don't want to battle with the debugger to figure out why.

Summary

- ASP.NET Core projects are created with the `dotnet new` command.
- There are templates to jumpstart popular project types and to create common project items.
- The `dotnet build` command compiles a project.
- The `dotnet run` command builds and executes a project.
- The `dotnet watch` command builds and executes a project, and performs hot reloading when changes are detected.
- Packages are added to a project with the `dotnet add package` command.
- Tool packages are installing using the `dotnet tool install` command.
- Client-side packages are managed with the `libman` tool package.